AJAY KADIYALA - Data Engineer

Follow me Here:

LinkedIn:
https://www.linkedin.com/in/ajay026/

Data Geeks Community:

https://lnkd.in/gU5NkCqi

SQL INTERVIEW QUESTIONS KIT FOR DATA ENGINEERS

SQL is the backbone of data engineering, essential for efficiently handling and querying large datasets. It's not just for databases anymore—SQL is key in tools like Apache Spark for big data processing. It’s a must-have in interviews, where SQL skills are often the first thing tested. Mastering SQL also helps you collaborate with analysts and scientists and forms a strong foundation for learning other data tools. In short, SQL is a core skill every data engineer needs to succeed.

This document contains 150+ SQL interview questions focusing on medium to advanced topics. This also Contains Recently asked questions in my interviews, and Scenario based questions. I have provided the Solution with Explanation to help you understand SQL concepts that are crucial for data engineering roles and to ace your interviews.

Table of Contents:

Complex Join Optimization

Question: You have three tables: Customers, Orders, and Payments. Write a query to find all customers who have made an order but have not completed a payment in the last 30 days.

Solution:

```
SELECT c.customer_id, c.name
FROM Customers c
JOIN Orders o ON c.customer_id = o.customer_id
LEFT JOIN Payments p ON o.order_id = p.order_id
AND p.payment_date > NOW() - INTERVAL 30 DAY
WHERE p.order_id IS NULL;
```

Explanation:

This query uses a LEFT JOIN and checks for null values in the Payments table to find customers with incomplete payments.

Window Functions in Analytical Queries

Question: For each product sold, calculate the running total of the number of sales made over time, ordered by sale date.

Solution:

```
SELECT product_id, sale_date, sales,
    SUM(sales) OVER (PARTITION BY product_id ORDER BY
sale_date) AS running_total
FROM Sales;
```

Explanation:

The window function SUM() calculates a running total within each product partition.

Time-Based Performance Data Analysis

Question: You are given a table Service_Requests with columns request_id, created_at, and resolved_at. Write a query to calculate the average time taken to resolve a request (in hours), excluding weekends.

Solution:

```
WITH request_durations AS (
  SELECT request_id,
       EXTRACT(EPOCH FROM (resolved_at - created_at))/3600
AS hours_taken
  FROM Service_Requests
  WHERE EXTRACT(DOW FROM created_at) BETWEEN 1 AND
5
)
SELECT AVG(hours_taken) AS avg_resolution_time
FROM request_durations;
```

Explanation:

This query filters out weekends and calculates the average time to resolve a request in hours.

Identifying Frequent Buyers

Question: Given a Purchases table, find customers who have made more than 5 purchases in the last 6 months, along with their total spend.

Solution:

```
SELECT customer_id, COUNT(*) AS purchase_count,
SUM(amount) AS total_spent
```

```
FROM Purchases
WHERE purchase_date > NOW() - INTERVAL 6 MONTH
GROUP BY customer_id
HAVING COUNT(*) > 5;
```

Explanation:

This query identifies frequent buyers and uses HAVING to filter customers based on their purchase count.

Finding Missing Values

Question: You have a table of student grades. Write a query to find students who do not have grades for all subjects.

Solution:

```
SELECT student_id
FROM Student_Grades
GROUP BY student_id
HAVING COUNT(DISTINCT subject_id) < (SELECT COUNT(*)
FROM Subjects);
```

Explanation:

This query checks for students whose count of distinct subjects is less than the total subject count.

Top N Employees by Sales

Question: Write a query to find the top 3 employees with the highest sales in the last year.

Solution:

```
SELECT employee_id, SUM(sales_amount) AS total_sales
FROM Sales
```

```
WHERE sale_date >= DATEADD(YEAR, -1, GETDATE())
GROUP BY employee_id
ORDER BY total_sales DESC
LIMIT 3;
```

Explanation:

This query sums sales for each employee and uses LIMIT to return the top three.

Calculating Percentiles

Question: How would you write a query to find the 90th percentile of sales from a Sales table?

Solution:

```
SELECT PERCENTILE_CONT(0.90) WITHIN GROUP (ORDER BY sales_amount) AS p90_sales
FROM Sales;
```

Explanation:

This uses the PERCENTILE_CONT function to calculate the 90th percentile of the sales.

Handling Nulls in Aggregations

Question: How can you write a query that sums the total sales but treats null values as zero?

Solution:

```
SELECT SUM(COALESCE(sales_amount, 0)) AS total_sales
FROM Sales;
```

Explanation:

The COALESCE function replaces null values with zero in the sum.

Date Differences in SQL

Question: Write a query to find the average time taken between order creation and delivery.

Solution:

```
SELECT AVG(DATEDIFF(day, order_date, delivery_date)) AS avg_delivery_time
FROM Orders;
```

Explanation:

DATEDIFF calculates the difference in days, and AVG returns the average time.

Dynamic SQL for Reporting

Question: Write a dynamic SQL query to generate a report of sales per month for a given year.

Solution:

```
DECLARE @Year INT = 2023;
EXEC('SELECT MONTH(sale_date) AS Month,
SUM(sales_amount) AS TotalSales
FROM Sales
WHERE YEAR(sale_date) = ' + CAST(@Year AS VARCHAR) + '
GROUP BY MONTH(sale_date);');
```

Explanation:

Dynamic SQL is constructed and executed to report monthly sales for the specified year.

Query Performance Optimization Techniques

Question: How can you identify and optimize slow-running queries in SQL?

Solution:

```
-- Use the EXPLAIN command to analyze query execution plans.
EXPLAIN SELECT * FROM Orders WHERE customer_id = 123;

-- Add appropriate indexes to improve query speed
CREATE INDEX idx_customer_id ON Orders (customer_id);

-- Optimize joins and reduce the number of rows processed.
SELECT o.order_id, c.customer_name
FROM Orders o
JOIN Customers c ON o.customer_id = c.customer_id
WHERE c.region = 'North';
```

Explanation:

You can optimize slow-running queries by analyzing execution plans, adding indexes, and optimizing joins.

Temporary vs. Permanent Tables

Question: What are the differences between temporary and permanent tables, and when would you use each?

Solution:

```
-- Temporary Table (Exists for the session)
CREATE TEMPORARY TABLE temp_orders AS
SELECT * FROM Orders;
```

```
-- Permanent Table (Exists indefinitely until dropped)
CREATE TABLE permanent_orders AS
SELECT * FROM Orders;
```

Explanation:

Temporary tables are used for session-specific data, while permanent tables are used for long-term data storage.

Full-Text Search Capabilities

Question: How do you implement full-text search on a column in SQL?

Solution:

```
-- Create a full-text index on a column
CREATE FULLTEXT INDEX ON Products(product_description);

-- Use the CONTAINS function to search for a keyword
SELECT * FROM Products WHERE
CONTAINS(product_description, 'laptop');
```

Explanation:

Full-text indexes and the CONTAINS function allow for advanced text searching in SQL databases.

Working with Hierarchical Data

Question: How do you query hierarchical data stored in a self-referencing table?

Solution:

```
WITH Hierarchy AS (
   SELECT employee_id, manager_id, name, 0 AS Level
```

```
    FROM Employees
    WHERE manager_id IS NULL
    UNION ALL
    SELECT e.employee_id, e.manager_id, e.name, h.Level + 1
    FROM Employees e
    INNER JOIN Hierarchy h ON e.manager_id = h.employee_id
)
SELECT * FROM Hierarchy;
```

Explanation:

This recursive CTE allows you to retrieve hierarchical data, starting from top-level managers and finding all related employees.

Understanding and Implementing Partitioning

Question: Explain how partitioning works in SQL and provide an example of partitioning a table by date.

Solution:

```
CREATE TABLE Sales (
    sale_id INT,
    sale_date DATE,
    amount DECIMAL(10, 2)
) PARTITION BY RANGE (YEAR(sale_date)) (
    PARTITION p2022 VALUES LESS THAN (2023),
    PARTITION p2023 VALUES LESS THAN (2024)
);
```

Explanation:

Partitioning helps manage large datasets by dividing them into smaller, more manageable pieces, such as by date ranges.

Handling Concurrency in Transactions

Question: How would you manage concurrency issues in SQL, such as the lost update problem?

Solution:

```
-- Set transaction isolation level to SERIALIZABLE
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;
-- Perform the update or insert operation
COMMIT;
```

Explanation:

Setting the transaction isolation level to SERIALIZABLE ensures that transactions are executed in a way that prevents concurrency issues.

Calculating Percentiles in SQL

Question: Write a query to calculate the 90th percentile of sales amounts.

Solution:

```
SELECT PERCENTILE_CONT(0.90) WITHIN GROUP (ORDER BY sales_amount) AS p90_sales
FROM Sales;
```

Explanation:

This query uses the PERCENTILE_CONT function to calculate the 90th percentile of the sales amounts.

Dynamic SQL Execution

Question: Write a dynamic SQL query that generates a report of sales by region for a given year.

Solution:

```
DECLARE @Year INT = 2023;
DECLARE @SQL NVARCHAR(MAX);

SET @SQL = 'SELECT region, SUM(sales_amount) AS total_sales
        FROM Sales WHERE YEAR(sale_date) = ' + CAST(@Year
AS NVARCHAR) + '
        GROUP BY region;';

EXEC sp_executesql @SQL;
```

Explanation:

Dynamic SQL allows for constructing and executing SQL statements based on variables, such as the year in this example.

Window Functions for Running Totals

Question: Write a query to calculate a running total of sales for each product.

Solution:

```
SELECT product_id, sale_date, sales_amount,
    SUM(sales_amount) OVER (PARTITION BY product_id
```

```
ORDER BY sale_date) AS running_total
FROM Sales;
```

Explanation:

This query uses the SUM() window function to calculate a running total for each product, partitioned by product_id.

Handling NULL Values in SQL

Question: How can you handle NULL values in a SUM operation to ensure they are treated as zero?

Solution:

```
SELECT SUM(COALESCE(sales_amount, 0)) AS total_sales
FROM Sales;
```

Explanation:

The COALESCE function is used to replace NULL values with 0 in the SUM operation to avoid incorrect totals.

Identifying Duplicates in SQL

Question: Write a query to identify duplicate rows in a table based on a combination of columns.

Solution:

```
SELECT customer_id, order_id, COUNT(*)
FROM Orders
GROUP BY customer_id, order_id
HAVING COUNT(*) > 1;
```

Explanation:

This query uses GROUP BY and HAVING to find duplicate records where customer_id and order_id are repeated.

Recursive CTE for Hierarchies

Question: How do you write a recursive CTE to navigate a hierarchy of departments?

Solution:

```
WITH DepartmentHierarchy AS (
   SELECT department_id, parent_department_id,
department_name, 0 AS Level
   FROM Departments
   WHERE parent_department_id IS NULL
   UNION ALL
   SELECT d.department_id, d.parent_department_id,
d.department_name, dh.Level + 1
   FROM Departments d
   INNER JOIN DepartmentHierarchy dh ON
d.parent_department_id = dh.department_id
)
SELECT * FROM DepartmentHierarchy;
```

Explanation:

This recursive CTE allows you to retrieve and navigate a hierarchy of departments by finding all related sub-departments.

Recursive Common Table Expressions

Question: Write a recursive CTE to find all employees in an organization and their respective managers, given an Employees table.

Solution:

```
WITH EmployeeCTE AS (
   SELECT employee_id, manager_id, name
   FROM Employees
   WHERE manager_id IS NULL
   UNION ALL
   SELECT e.employee_id, e.manager_id, e.name
   FROM Employees e
   INNER JOIN EmployeeCTE cte ON e.manager_id =
cte.employee_id
)
SELECT * FROM EmployeeCTE;
```

Explanation:

This recursive CTE starts with top-level managers and recursively joins to find all employees under each manager.

Data Transformation with CASE

Question: Write a query that categorizes sales amounts into different ranges using the CASE statement.

Solution:

```
SELECT sale_amount,
      CASE
         WHEN sale_amount < 100 THEN 'Low'
```

```
        WHEN sale_amount BETWEEN 100 AND 500 THEN
'Medium'
        ELSE 'High'
    END AS sale_category
FROM Sales;
```

Explanation:

This query uses the CASE statement to categorize sales amounts into 'Low', 'Medium', and 'High'.

Using JSON Functions in SQL

Question: Given a JSON column named `data` in a table, write a query to extract a specific value from the JSON object.

Solution:

```
SELECT data->>'$.name' AS customer_name
FROM Customers;
```

Explanation:

This query extracts the `name` field from a JSON column in the Customers table.

Ranking Functions and Their Applications

Question: Write a query to rank employees based on their sales performance within each department.

Solution:

```
SELECT employee_id, department_id, sales,
    RANK() OVER (PARTITION BY department_id ORDER BY
sales DESC) AS sales_rank
FROM EmployeeSales;
```

Explanation:

The RANK() function assigns a rank to each employee based on their sales, partitioned by department.

Creating Pivot Tables Using SQL

Question: Write a query to pivot sales data to show the total sales for each product by month.

Solution:

```
SELECT product_id,
    SUM(CASE WHEN MONTH(sale_date) = 1 THEN
sales_amount ELSE 0 END) AS January,
    SUM(CASE WHEN MONTH(sale_date) = 2 THEN
sales_amount ELSE 0 END) AS February,
    SUM(CASE WHEN MONTH(sale_date) = 3 THEN
sales_amount ELSE 0 END) AS March
FROM Sales
GROUP BY product_id;
```

Explanation:

This query pivots sales data, aggregating total sales for each product by month using conditional aggregation.

Advanced Use of GROUP BY with ROLLUP and CUBE

Question: Write a query to calculate total sales with subtotals for each product and grand total.

Solution:

```
SELECT product_id, SUM(sales_amount) AS total_sales
FROM Sales
GROUP BY ROLLUP(product_id);
```

Explanation:

The ROLLUP operator provides subtotal and grand total rows in the result set.

Error Handling in SQL Transactions

Question: How would you write a SQL transaction that rolls back if an error occurs?

Solution:

```
BEGIN TRY
   BEGIN TRANSACTION;
   -- SQL operations
   INSERT INTO Orders (order_id, customer_id) VALUES (1, 1);
   -- Simulate error
   INSERT INTO Orders (order_id, customer_id) VALUES (NULL,
NULL);
   COMMIT TRANSACTION;
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION;
   PRINT ERROR_MESSAGE();
END CATCH;
```

Explanation:

This transaction block uses TRY...CATCH for error handling. If an error occurs, it rolls back all operations.

Implementing Triggers and Their Use Cases

Question: Write a trigger that automatically updates a timestamp column when a record is modified.

Solution:

```
CREATE TRIGGER UpdateTimestamp
ON Employees
AFTER UPDATE
AS
BEGIN
   UPDATE Employees
   SET last_modified = GETDATE()
   WHERE employee_id IN (SELECT employee_id FROM
inserted);
END;
```

Explanation:

This trigger updates the `last_modified` column with the current date and time whenever an employee record is updated.

Cross Joins and Their Implications

Question: What is a cross join, and provide a query example that demonstrates its use?

Solution:

```
SELECT a.name, b.product_name
FROM Customers a
CROSS JOIN Products b;
```

Explanation:

A cross join produces a Cartesian product, combining every row from one table with every row from another.

Using MERGE Statements for Upserts

Question: Write a query to use the MERGE statement for updating existing records or inserting new ones.

Solution:

```
MERGE INTO TargetTable AS target
USING SourceTable AS source
ON target.id = source.id
WHEN MATCHED THEN
   UPDATE SET target.value = source.value
WHEN NOT MATCHED THEN
   INSERT (id, value) VALUES (source.id, source.value);
```

Explanation:

The MERGE statement combines insert and update operations based on matching criteria.

Handling NULL Values in Conditional Aggregations

Question: Write a query that counts the number of orders placed by customers, treating NULL customer IDs as 'Unknown'.

Solution:

```
SELECT COALESCE(customer_id, 'Unknown') AS
customer_identifier,
COUNT(order_id) AS total_orders
FROM Orders
GROUP BY COALESCE(customer_id, 'Unknown');
```

Explanation:

This query uses the COALESCE function to treat NULL customer IDs as 'Unknown' and counts the orders per customer.

Combining Data from Multiple Sources

Question: How can you combine data from two different tables that have similar structures, such as Orders2022 and Orders2023?

Solution:

```
SELECT * FROM Orders2022
UNION ALL
SELECT * FROM Orders2023;
```

Explanation:

This query combines rows from two tables using UNION ALL, which includes all rows without eliminating duplicates.

Updating Data with Conditions

Question: Write a query to update the status of all orders that have been shipped but not delivered.

Solution:

```
UPDATE Orders
SET status = 'In Transit'
WHERE shipped_date IS NOT NULL
AND delivered_date IS NULL;
```

Explanation:

This query updates the order status to 'In Transit' for orders that have been shipped but not yet delivered.

Ranking with DENSE_RANK

Question: Write a query to rank employees by their total sales, without skipping rank numbers for ties.

Solution:

```
SELECT employee_id, total_sales,
DENSE_RANK() OVER (ORDER BY total_sales DESC) AS
sales_rank
FROM EmployeeSales;
```

Explanation:

DENSE_RANK ranks employees by total sales without gaps in rank numbers for tied values.

Using EXISTS for Subqueries

Question: Write a query to find customers who have placed at least one order in the last month.

Solution:

```
SELECT customer_id, customer_name
FROM Customers c
WHERE EXISTS (
   SELECT 1 FROM Orders o
   WHERE o.customer_id = c.customer_id
   AND o.order_date > DATEADD(MONTH, -1, GETDATE())
);
```

Explanation:

The EXISTS clause checks if a customer has placed at least one order in the last month, returning only those customers.

Recursive Queries for Tree Structures

Question: How would you write a recursive query to retrieve all managers and their subordinates from an Employees table?

Solution:

```
WITH EmployeeHierarchy AS (
   SELECT employee_id, manager_id, name, 0 AS Level
   FROM Employees
   WHERE manager_id IS NULL
   UNION ALL
   SELECT e.employee_id, e.manager_id, e.name, eh.Level + 1
   FROM Employees e
   INNER JOIN EmployeeHierarchy eh ON e.manager_id =
eh.employee_id
)
SELECT * FROM EmployeeHierarchy;
```

Explanation:

This recursive query retrieves all employees and their respective managers, creating a hierarchical tree structure.

Efficient Data Filtering with Indexes

Question: How can you improve the performance of a query that filters data by customer_id in a large Orders table?

Solution:

```
CREATE INDEX idx_customer_id ON Orders(customer_id);

-- Query utilizing the index for faster lookups
SELECT * FROM Orders WHERE customer_id = 123;
```

Explanation:

Creating an index on the customer_id column improves the performance of queries filtering by that column.

Using LAG and LEAD Window Functions

Question: Write a query to compare each employee's sales with their previous month's sales.

Solution:

```
SELECT employee_id, sale_month, sales_amount,
    LAG(sales_amount) OVER (PARTITION BY employee_id
ORDER BY sale_month) AS previous_month_sales
FROM EmployeeSales;
```

Explanation:

The LAG function allows you to access the previous month's sales for each employee in the result set.

Handling Aggregations with DISTINCT

Question: Write a query to count the number of unique customers who placed orders in the last year.

Solution:

```
SELECT COUNT(DISTINCT customer_id) AS unique_customers
FROM Orders
WHERE order_date > DATEADD(YEAR, -1, GETDATE());
```

Explanation:

The DISTINCT keyword ensures that only unique customer_ids are counted in the result.

Creating Custom Aggregates with CASE

Question: Write a query to calculate the total sales for each product category, with a separate total for 'Electronics'.

Solution:

```
SELECT category,
    SUM(CASE WHEN category = 'Electronics' THEN
sales_amount ELSE 0 END) AS electronics_sales,
    SUM(sales_amount) AS total_sales
FROM Sales
GROUP BY category;
```

Explanation:

This query uses the CASE statement to calculate sales specifically for 'Electronics' and overall sales for each category.

Combining Multiple Aggregations in a Single Query

Question: Write a query to calculate the average, minimum, and maximum sales amount for each region.

Solution:

```
SELECT region,
    AVG(sales_amount) AS average_sales,
    MIN(sales_amount) AS min_sales,
    MAX(sales_amount) AS max_sales
FROM Sales
GROUP BY region;
```

Explanation:

This query combines multiple aggregation functions (AVG, MIN, MAX) to provide various statistics for each region.

Efficient Pagination with OFFSET and FETCH

Question: Write a query to return the second page of results from an Orders table, assuming each page shows 10 results.

Solution:

```
SELECT * FROM Orders
ORDER BY order_date
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY;
```

Explanation:

This query uses OFFSET and FETCH to skip the first 10 rows and return the next 10 rows, simulating pagination.

Working with Self-Joins

Question: Write a query to find all employees who share the same manager in an Employees table.

Solution:

```
SELECT e1.employee_id, e1.name AS employee_name, e2.name
AS manager_name
FROM Employees e1
JOIN Employees e2 ON e1.manager_id = e2.employee_id;
```

Explanation:

A self-join is used to match employees with their managers by joining the Employees table to itself.

Date and Time Calculations

Question: Write a query to calculate the number of days between an order date and a delivery date.

Solution:

```
SELECT order_id, DATEDIFF(day, order_date, delivery_date) AS
days_to_deliver
FROM Orders
WHERE delivery_date IS NOT NULL;
```

Explanation:

The DATEDIFF function calculates the difference in days between the order_date and delivery_date for each order.

Detecting Gaps in Sequences

Question: How can you write a query to detect missing invoice numbers in a sequence of invoices?

Solution:

```
WITH OrderedInvoices AS (
   SELECT invoice_id, LAG(invoice_id) OVER (ORDER BY
invoice_id) AS prev_invoice
   FROM Invoices
)
SELECT invoice_id
FROM OrderedInvoices
WHERE invoice_id <> prev_invoice + 1;
```

Explanation:

This query uses the LAG window function to compare each invoice_id with the previous one, detecting gaps in the sequence.

Explain the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN with examples.

Answer:

- **INNER JOIN** returns records that have matching values in both tables.
- **LEFT JOIN** returns all records from the left table and matching records from the right.
- **RIGHT JOIN** returns all records from the right table and matching records from the left.
- **FULL JOIN** returns all records when there is a match in either table.

```
Example:

SELECT a.column, b.column

FROM table_a a

INNER JOIN table_b b ON a.id = b.id;
```

How would you find duplicate rows in a table?

Answer: Use GROUP BY and HAVING to count duplicates.

Solution:

```
SELECT column1, COUNT(*)

FROM table

GROUP BY column1
```

```
HAVING COUNT(*) > 1;
```

What's the difference between RANK() and DENSE_RANK() in SQL?

Answer:

- **RANK()** gives gaps in ranking after ties.
- **DENSE_RANK()** assigns consecutive ranks without gaps.

Example:

Solution:

```
SELECT column, RANK() OVER (ORDER BY column DESC) AS rank
FROM table;
```

How would you remove duplicate rows in SQL?

Answer: Use ROW_NUMBER() with a Common Table Expression (CTE) to delete duplicates.

Solution:

```
WITH cte AS (
   SELECT column, ROW_NUMBER() OVER(PARTITION BY column ORDER BY column) AS row_num
   FROM table
)
```

```
DELETE FROM cte WHERE row_num > 1;
```

Explain window functions and how you would use them in SQL.

Answer: Window functions perform calculations across a set of rows related to the current row, like RANK(), SUM(), AVG(), and ROW_NUMBER(). They are useful for calculations without collapsing rows.

Solution:

```
SELECT column, SUM(value) OVER(PARTITION BY category) AS sum_value
FROM table;
```

How would you fetch only even-numbered rows from a table?

Answer:

Solution:

```
SELECT *
FROM (
  SELECT *, ROW_NUMBER() OVER (ORDER BY column) AS row_num
  FROM table
) AS subquery
WHERE row_num % 2 = 0;
```

What is the purpose of the EXPLAIN statement in SQL?

Answer: EXPLAIN analyzes the query execution plan, showing how tables are accessed, indices used, and estimated costs, allowing you to optimize queries.

Explain a scenario where you'd use a SELF JOIN.

Answer: Use a SELF JOIN when you need to compare rows within the same table, such as finding employee-manager relationships within an employee table.

Solution:

```
SELECT a.employee_id, b.employee_id AS manager_id
FROM employees a
JOIN employees b ON a.manager_id = b.employee_id;
```

How can you optimize a query with multiple joins?

Answer: Index columns involved in joins, filter data early, and order joins so smaller tables are joined first. Use EXPLAIN to check performance.

Describe the difference between WHERE and HAVING.

Answer: WHERE filters rows before aggregation, while HAVING filters after aggregation. Use WHERE for raw data and HAVING for aggregate functions.

How would you calculate the cumulative sum in SQL?

Answer:

Solution:

```
SELECT column, SUM(value) OVER (ORDER BY column ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cumulative_sum
FROM table;
```

What is the difference between UNION and UNION ALL?

Answer: UNION removes duplicates, while UNION ALL keeps all results, including duplicates. UNION ALL is faster for larger datasets.

How would you delete rows that are duplicates but keep the first occurrence?

Answer: Use ROW_NUMBER() with a CTE to assign row numbers and delete duplicates.

Solution:

```
WITH cte AS (
   SELECT *, ROW_NUMBER() OVER(PARTITION BY column ORDER BY column) AS row_num
   FROM table
```

```
)
DELETE FROM cte WHERE row_num > 1;
```

Explain a correlated subquery and give an example.

Answer: A correlated subquery is a subquery that depends on the outer query for its values. Example:

Solution:

```
SELECT name
FROM employees e1
WHERE salary > (
   SELECT AVG(salary)
   FROM employees e2
   WHERE e1.department_id = e2.department_id
);
```

What is an index, and how does it improve query performance?

Answer: An index is a data structure that speeds up data retrieval by reducing the amount of data scanned. It acts as a pointer to data rows based on indexed columns.

What's the difference between a PRIMARY KEY and a UNIQUE constraint?

Answer: Both enforce uniqueness, but PRIMARY KEY also doesn't allow NULL values and uniquely identifies each row, while UNIQUE allows one NULL.

How do you find the nth highest salary in SQL?

Answer:

Solution:

```
SELECT DISTINCT salary
FROM employees
ORDER BY salary DESC
LIMIT 1 OFFSET n-1;
```

What is a TRIGGER and when would you use it?

Answer: A TRIGGER automatically executes predefined actions in response to database events (like insert, update, delete). It's useful for logging changes or enforcing rules.

How would you update records in one table based on values in another table?

Answer: Use JOIN in the UPDATE statement.

Solution:

```
UPDATE table1
```

```
SET column = table2.value
FROM table2
WHERE table1.id = table2.id;
```

Explain the COALESCE function and give an example.

Answer: COALESCE returns the first non-null value among its arguments, useful for handling missing values.

Solution:

```
SELECT COALESCE(column1, column2, 0) AS result FROM table;
```

How would you remove rows with NULL values in a specific column?

Answer:

Solution:

```
DELETE FROM table WHERE column IS NULL;
```

How would you list all unique pairs of columns from the same table?

Answer:

Solution:

```
SELECT DISTINCT a.column1, b.column1
FROM table a
```

```
JOIN table b ON a.column1 < b.column1;
```

Explain the GROUP BY clause and when you would use it.

Answer: GROUP BY groups rows sharing a common field, used with aggregate functions like SUM, COUNT, AVG.

24. What is ACID in database transactions?

Answer: ACID stands for Atomicity, Consistency, Isolation, Durability. It ensures reliable transactions, essential for maintaining data integrity.

25. How would you calculate the moving average in SQL?

Answer:

Solution:

```
SELECT column, AVG(value) OVER(ORDER BY date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS moving_avg
FROM table;
```

Explain what a cross join is and when you might use it.

Answer: A cross join returns the Cartesian product of two tables. It's rarely used unless you need all possible combinations of two sets.

What is the difference between DELETE and TRUNCATE?

Answer: DELETE removes rows one by one, can have a WHERE clause, and can be rolled back. TRUNCATE removes all rows quickly, can't be rolled back, and resets identity columns.

How do you handle NULL values when joining tables?

Answer: Use LEFT JOIN to retain rows with NULL in one table, or use IS NULL or COALESCE to manage NULL values.

How would you select records in one table that don't exist in another table?

Answer: Use LEFT JOIN or NOT EXISTS.

Solution:

```
SELECT a.*
FROM table_a a
LEFT JOIN table_b b ON a.id = b.id
WHERE b.id IS NULL;
```

How would you pivot data in SQL?

Answer: Use conditional aggregation with CASE or use the PIVOT function if supported.

Solution:

```
SELECT category, SUM(CASE WHEN month = 'January' THEN value END) AS January

FROM sales

GROUP BY category;
```

How would you normalize a table, and why is it important?

Answer: Normalization removes redundancy by structuring data across tables, enforcing dependencies, and improving consistency.

How would you denormalize data for performance?

Answer: Combine tables to reduce joins, allowing faster reads, often used in reporting databases.

How would you create an index, and when should you avoid it?

Answer: Use CREATE INDEX for frequently searched columns but avoid over-indexing on tables with frequent writes, as it slows down insertions.

Explain the use of IFNULL and NULLIF.

Answer: IFNULL replaces NULL with a default value, while NULLIF returns NULL if two expressions are equal.

How do you find the first and last records in a grouped dataset?

Answer:

Solution:

```
SELECT category, FIRST_VALUE(value) OVER (PARTITION BY category ORDER BY date) AS first_value
```

How would you write a recursive query?

Answer: Use Common Table Expressions (CTEs) for recursion.

Solution:

```
WITH RECURSIVE cte AS (
   SELECT column FROM table
   UNION ALL
   SELECT column FROM table WHERE condition
)
```

Explain a CASE statement and give an example.

Answer:

Solution:

```
SELECT name,
```

```
  CASE
    WHEN score > 90 THEN 'A'
    WHEN score > 80 THEN 'B'
    ELSE 'C'
  END AS grade
FROM students;
```

What are SQL constraints, and why are they important?

Answer: Constraints enforce rules on data, such as PRIMARY KEY, FOREIGN KEY, and CHECK, to maintain integrity.

What is a materialized view?

Answer: A materialized view stores the results of a query physically for fast access, unlike regular views which are computed on demand.

How would you schedule an SQL job to run periodically?

Answer: Use database schedulers, such as SQL Server Agent or cron jobs.

What is the purpose of the LIMIT and OFFSET clauses?

Answer: LIMIT restricts result rows, while OFFSET skips a number of rows, useful for pagination.

Explain the difference between VARCHAR and CHAR.

Answer: CHAR has a fixed length, while VARCHAR is variable-length, saving space for shorter values.

How would you handle transactional deadlocks in SQL?

Answer: Use retry logic, lower isolation levels, and index tables appropriately to reduce deadlocks.

What is a stored procedure, and why use one?

Answer: A stored procedure is a saved SQL script that can be reused, which helps with performance, code organization, and security.

How would you handle time zone conversions in SQL?

Answer: Use AT TIME ZONE or equivalent functions to convert between time zones.

Explain the use of PARTITION BY and ORDER BY in window functions.

Answer: PARTITION BY groups rows, and ORDER BY sets the order within each partition, crucial for calculations like ranking and cumulative sums.

How do you measure query performance in SQL?

Answer: Use execution plans, query runtime, index usage stats, and track CPU/memory usage.

What is a temp table, and when would you use it?

Answer: Temporary tables store intermediate results, ideal for complex queries and session-specific data.

How would you handle schema changes in a production database?

Answer: Test changes in a staging environment, backup data, use ALTER TABLE carefully, and communicate downtime if needed.

What is a CTE, and how does it differ from a subquery?

Answer: A CTE is a temporary result set within a query for better readability and reusability, while a subquery is a nested query directly within another query.

How would you identify customers who made a purchase last year but not this year?

Answer: Use a LEFT JOIN and filter out the records with no matches from this year.

Solution:

```
SELECT last_year.customer_id

FROM purchases last_year

LEFT JOIN purchases this_year

ON last_year.customer_id = this_year.customer_id AND
this_year.year = 2023

WHERE last_year.year = 2022 AND this_year.customer_id IS
NULL;
```

Given a sales table, how do you find the top 3 sales representatives based on total sales?

Answer: Use ORDER BY and LIMIT to rank the sales totals.

Solution:

```
SELECT representative, SUM(sales) AS total_sales

FROM sales

GROUP BY representative

ORDER BY total_sales DESC

LIMIT 3;
```

How would you find customers who've bought all products in a given product list?

Answer: Use a HAVING clause to count the distinct products purchased by each customer and compare it to the list's count.

Solution:

```
SELECT customer_id
FROM purchases
WHERE product_id IN (1, 2, 3) -- assuming these are the product IDs
GROUP BY customer_id
HAVING COUNT(DISTINCT product_id) = 3;
```

How do you identify consecutive absences in attendance data?

Answer: Use LAG or LEAD to compare current and previous records.

Solution:

```
SELECT employee_id, date,
   LAG(status) OVER (PARTITION BY employee_id ORDER BY date) AS prev_status
FROM attendance
WHERE status = 'absent';
```

How would you calculate the percentage growth in sales month over month?

Answer: Use LAG to access previous month's sales and calculate the growth percentage.

Solution:

```
SELECT month, sales,
    (sales - LAG(sales) OVER (ORDER BY month)) / LAG(sales) OVER (ORDER BY month) * 100 AS growth_percentage
FROM sales_data;
```

How can you identify orders that have been shipped but not yet billed?

Answer: Use a LEFT JOIN between shipped orders and billing records.

Solution:

```
SELECT orders.order_id
FROM orders
LEFT JOIN billing ON orders.order_id = billing.order_id
WHERE orders.status = 'shipped' AND billing.order_id IS NULL;
```

Given transaction data, how would you find the day with the highest transactions?

Answer: Group by day and order by transaction count.

Solution:

```
SELECT day, COUNT(*) AS transaction_count
FROM transactions
GROUP BY day
ORDER BY transaction_count DESC
LIMIT 1;
```

How would you rank employees within departments based on their performance score?

Answer: Use RANK() with PARTITION BY department_id.

Solution:

```
SELECT employee_id, department_id, performance_score,
    RANK() OVER (PARTITION BY department_id ORDER BY performance_score DESC) AS rank
FROM employees;
```

How would you retrieve customers who placed exactly two orders?

Answer: Group by customer and use HAVING to filter.

Solution:

```
SELECT customer_id
FROM orders
GROUP BY customer_id
HAVING COUNT(order_id) = 2;
```

How do you find the first purchase made by each customer?

Answer: Use ROW_NUMBER() to rank purchases by date and filter by the first one.

Solution:

```
SELECT customer_id, purchase_date, amount
FROM (
   SELECT customer_id, purchase_date, amount,
        ROW_NUMBER() OVER (PARTITION BY customer_id
ORDER BY purchase_date) AS row_num
   FROM purchases
) AS first_purchases
WHERE row_num = 1;
```

How do you identify orders with a price change compared to the previous order?

Answer: Use LAG to compare the previous order's price.

Solution:

```
SELECT order_id, price, LAG(price) OVER (ORDER BY
order_date) AS prev_price

FROM orders

WHERE price != prev_price;
```

How would you find the average sales by month and year?

Answer: Group by both YEAR and MONTH functions.

Solution:

```
SELECT YEAR(sale_date) AS year, MONTH(sale_date) AS
month, AVG(sales) AS avg_sales

FROM sales

GROUP BY year, month;
```

How do you find customers who placed orders in two consecutive months?

Answer: Use LAG to check for consecutive orders.

Solution:

```
SELECT customer_id, MONTH(order_date) AS month,
YEAR(order_date) AS year

FROM (

  SELECT customer_id, order_date,
```

```
        DATEDIFF(MONTH, LAG(order_date) OVER (PARTITION
BY customer_id ORDER BY order_date), order_date) AS
month_gap
    FROM orders
) AS consecutive_orders
WHERE month_gap = 1;
```

How would you get the running total of sales over time?

Answer: Use SUM with OVER for cumulative totals.

Solution:

```
SELECT sale_date, sales,
      SUM(sales) OVER (ORDER BY sale_date) AS running_total
FROM sales;
```

How would you handle a situation where you need to delete duplicate records, keeping only the latest entry?

Answer: Use ROW_NUMBER() to identify duplicates based on timestamp and delete them.

Solution:

```
WITH deduped AS (
    SELECT *, ROW_NUMBER() OVER(PARTITION BY
unique_key ORDER BY timestamp DESC) AS row_num
```

```
   FROM records
)
DELETE FROM deduped WHERE row_num > 1;
```

My Interview Experience

How do you calculate the time difference between two timestamps in SQL?

Answer:

Solution:

```
SELECT TIMESTAMPDIFF(HOUR, start_time, end_time) AS hours_diff
FROM table;
```

How would you retrieve products with the same price across multiple orders?

Answer: Use a GROUP BY and HAVING with a price count check.

Solution:

```
SELECT product_id, price
FROM orders
GROUP BY product_id, price
```

```
HAVING COUNT(DISTINCT order_id) > 1;
```

How do you find all orders placed in the last 7 days?

Answer:

Solution:

```
SELECT *
FROM orders
WHERE order_date >= DATEADD(DAY, -7, GETDATE());
```

How would you retrieve the order with the maximum amount for each customer?

Answer: Use ROW_NUMBER() with partitioning.

Solution:

```
SELECT customer_id, order_id, amount
FROM (
   SELECT customer_id, order_id, amount,
        ROW_NUMBER() OVER (PARTITION BY customer_id
ORDER BY amount DESC) AS row_num
   FROM orders
) AS max_orders
WHERE row_num = 1;
```

How do you calculate monthly retention rates for customers?

Answer: Join customer tables on consecutive months to check if they returned.

Solution:

```
SELECT a.month, COUNT(DISTINCT a.customer_id) AS retained
FROM customers a
JOIN customers b ON a.customer_id = b.customer_id AND
a.month = b.month - 1;
```

How would you identify the best-selling product each month?

Answer: Use ROW_NUMBER() to rank products by sales each month.

Solution:

```
WITH monthly_sales AS (
  SELECT product_id, MONTH(sale_date) AS month,
SUM(quantity) AS total_quantity,
      ROW_NUMBER() OVER (PARTITION BY month ORDER
BY SUM(quantity) DESC) AS rank
  FROM sales
  GROUP BY product_id, MONTH(sale_date)
)
```

```
SELECT product_id, month, total_quantity
FROM monthly_sales
WHERE rank = 1;
```

How do you handle a case where you need to split data between weekends and weekdays?

Answer: Use CASE in a SELECT statement to label weekdays and weekends.

Solution:

```
SELECT sale_date,
    CASE WHEN WEEKDAY(sale_date) IN (5, 6) THEN 'Weekend' ELSE 'Weekday' END AS day_type
FROM sales;
```

How do you calculate the difference in sales between two periods?

Answer:

Solution:

```
SELECT year, month,
    sales - LAG(sales) OVER (ORDER BY year, month) AS sales_difference
FROM sales;
```

How would you calculate a cohort retention rate?

Answer: Group customers based on acquisition date and calculate retention by matching those who returned.

Solution:

```
SELECT cohort, COUNT(DISTINCT
retained_customers.customer_id) / COUNT(DISTINCT
initial_customers.customer_id) AS retention_rate

FROM customers initial_customers

LEFT JOIN customers retained_customers ON
initial_customers.customer_id =
retained_customers.customer_id;
```

How would you identify customers who haven't purchased in over 6 months?

Answer:

Solution:

```
SELECT customer_id

FROM purchases

WHERE DATEDIFF(MONTH, last_purchase_date, GETDATE()) >
6;
```

How would you find employees who joined within the last quarter?

Answer:

Solution:

```
SELECT employee_id
FROM employees
WHERE hire_date >= DATEADD(QUARTER, -1, GETDATE());
```

How do you retrieve the last three months' average sales per day?

Answer:

Solution:

```
SELECT AVG(sales) AS avg_sales, DATE(sale_date) AS date
FROM sales
WHERE sale_date >= DATEADD(MONTH, -3, GETDATE())
GROUP BY date;
```

How do you identify items in stock at multiple stores?

Answer: Use GROUP BY and HAVING to identify products in multiple stores.

Solution:

```
SELECT product_id
FROM inventory
GROUP BY product_id
```

```
HAVING COUNT(DISTINCT store_id) > 1;
```

How do you select all employees with higher salaries than their department average?

Answer:

Solution:

```
SELECT employee_id
FROM employees e
JOIN (SELECT department_id, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department_id) dept_avg
ON e.department_id = dept_avg.department_id
WHERE e.salary > dept_avg.avg_salary;
```

How would you identify users who placed their last order in December last year?

Answer:

Solution:

```
SELECT customer_id
FROM orders
```

```
WHERE MONTH(last_order_date) = 12 AND
YEAR(last_order_date) = YEAR(GETDATE()) - 1;
```

How would you calculate the total sales for each product category by year?

Answer:

Solution:

```
SELECT category, YEAR(sale_date) AS year, SUM(sales) AS
total_sales

FROM products

JOIN sales ON products.product_id = sales.product_id

GROUP BY category, year;
```

How do you retrieve products with the second-highest sales for each category?

Answer: Use DENSE_RANK() to rank sales and filter by the second position.

Solution:

```
WITH ranked_sales AS (

   SELECT category, product_id, SUM(sales) AS total_sales,

        DENSE_RANK() OVER (PARTITION BY category ORDER
BY SUM(sales) DESC) AS rank
```

```
   FROM sales
   GROUP BY category, product_id
)
SELECT category, product_id, total_sales
FROM ranked_sales
WHERE rank = 2;
```

How would you find the average order amount for the top 10% of orders?

Answer: Use NTILE() to rank orders into percentiles.

Solution:

```
WITH percentiles AS (
   SELECT order_id, amount, NTILE(10) OVER (ORDER BY amount DESC) AS percentile
   FROM orders
)
SELECT AVG(amount) AS top_10_percent_avg
FROM percentiles
WHERE percentile = 1;
```

How do you find orders that took longer than the average processing time?

Answer:

Solution:

```
```

How would you find customers with both high purchase frequency and high average order value?

Answer: Use a HAVING clause to filter based on both frequency and average value.

Solution:

```
SELECT customer_id, COUNT(order_id) AS order_count, AVG(order_amount) AS avg_order
FROM orders
GROUP BY customer_id
HAVING order_count > threshold AND avg_order > threshold_value;
```

How do you get the last three months' revenue and average revenue per month?

Answer:

Solution:

```
SELECT YEAR(sale_date) AS year, MONTH(sale_date) AS month, SUM(revenue) AS total_revenue, AVG(revenue) AS avg_monthly_revenue
FROM sales
```

```
WHERE sale_date >= DATEADD(MONTH, -3, GETDATE())
GROUP BY year, month;
```

How would you retrieve customers who've never made a purchase?

Answer:

Solution:

```
SELECT customer_id
FROM customers
LEFT JOIN orders ON customers.customer_id =
orders.customer_id
WHERE orders.customer_id IS NULL;
```

How would you check for salary discrepancies between employees in the same role?

Answer:

Solution:

```
SELECT role, MAX(salary) - MIN(salary) AS salary_discrepancy
FROM employees
GROUP BY role
HAVING salary_discrepancy > 0;
```

How do you calculate the average time between two events in an event log?

Answer: Use LAG to access the previous timestamp and calculate the time difference.

Solution:

```
SELECT event_id, timestamp,
    AVG(TIMESTAMPDIFF(SECOND, LAG(timestamp) OVER (ORDER BY timestamp), timestamp)) AS avg_time_diff
FROM events;
```

How would you find products that haven't been sold in the last six months?

Answer:

Solution:

```
SELECT product_id
FROM products
LEFT JOIN sales ON products.product_id = sales.product_id
WHERE sales.sale_date < DATEADD(MONTH, -6, GETDATE())
OR sales.product_id IS NULL;
```

How would you retrieve the highest daily sales for each month?

Answer:

Solution:

```
SELECT month, MAX(daily_sales) AS max_sales
FROM (
   SELECT DATE(sale_date) AS day, MONTH(sale_date) AS month, SUM(amount) AS daily_sales
   FROM sales
   GROUP BY day, month
) daily_totals
GROUP BY month;
```

How do you rank products within each category by total revenue?

Answer:

Solution:

```
SELECT product_id, category, SUM(revenue) AS total_revenue,
     RANK() OVER (PARTITION BY category ORDER BY SUM(revenue) DESC) AS category_rank
FROM products
GROUP BY product_id, category;
```

How would you handle outliers in a dataset for sales orders?

Answer: Use PERCENTILE_CONT to define acceptable thresholds for identifying outliers.

Solution:

```
SELECT *

FROM orders

WHERE amount < PERCENTILE_CONT(0.05) WITHIN GROUP
(ORDER BY amount)

OR amount > PERCENTILE_CONT(0.95) WITHIN GROUP
(ORDER BY amount);
```

How would you find customers whose total spending is within the top 20% of all customers?

Answer:

Solution:

```
WITH spending_ranks AS (

   SELECT customer_id, SUM(amount) AS total_spent,

        NTILE(5) OVER (ORDER BY SUM(amount) DESC) AS
top_quintile

   FROM orders

   GROUP BY customer_id

)

SELECT customer_id, total_spent

FROM spending_ranks

WHERE top_quintile = 1;
```

How do you select all orders with their most recent status?

Answer:

Solution:

```
WITH latest_status AS (
   SELECT order_id, status, ROW_NUMBER() OVER (PARTITION BY order_id ORDER BY status_date DESC) AS row_num
   FROM order_statuses
)
SELECT order_id, status
FROM latest_status
WHERE row_num = 1;
```

How would you identify customers with irregular ordering patterns (e.g., high variance in order frequency)?

Answer:

Solution:

```
SELECT customer_id, STDDEV(order_interval) AS interval_variance
FROM (
```

```
   SELECT customer_id, DATEDIFF(day, LAG(order_date) OVER
(PARTITION BY customer_id ORDER BY order_date),
order_date) AS order_interval
   FROM orders
) intervals
GROUP BY customer_id
HAVING interval_variance > threshold;
```

How do you retrieve customers who haven't ordered in the current year but did in previous years?

Answer:

Solution:

```
SELECT customer_id
FROM orders
GROUP BY customer_id
HAVING MAX(YEAR(order_date)) < YEAR(GETDATE());
```

How would you calculate the month-over-month growth rate in SQL?

Answer:

Solution:

```
SELECT month, revenue,
```

```
    (revenue - LAG(revenue) OVER (ORDER BY month)) /
NULLIF(LAG(revenue) OVER (ORDER BY month), 0) * 100 AS
growth_rate

FROM monthly_revenue;
```

How do you identify products that have been ordered together frequently?

Answer:

Solution:

```
SELECT a.product_id, b.product_id, COUNT(*) AS
times_ordered_together

FROM order_items a

JOIN order_items b ON a.order_id = b.order_id AND
a.product_id < b.product_id

GROUP BY a.product_id, b.product_id

HAVING times_ordered_together > threshold;
```

How would you find the most popular time slots for orders in a day?

Answer:

Solution:

```
SELECT HOUR(order_time) AS hour, COUNT(*) AS order_count

FROM orders
```

```
GROUP BY hour
ORDER BY order_count DESC
LIMIT 3;
```

Best Practices for Writing SQL Code…

- Use Proper Indexing
- Write Clear and Readable Queries
- **Avoid Using SELECT ***
- Optimize Joins with the Correct Keys
- Use Proper Data Types
- Normalize Your Database Design
- Avoid Redundant Data
- Use Transactions Appropriately
- Leverage Query Caching
- Use LIMIT to Fetch Required Rows
- Optimize WHERE Clauses
- Avoid Using Cursors When Not Necessary
- Partition Large Tables
- Regularly Monitor and Tune Performance
- Use Window Functions for Advanced Analytics

In Order to help you Please Find SQL RoadMap here for FREE with resources…..

https://www.linkedin.com/posts/ajay026_sqloptimizations-activity-7045023960769466368-VFKZ?utm_source=share&utm_medium=member_desktop

Top SQL Commands you should learn……

https://www.linkedin.com/posts/ajay026_sql-commands-activity-7039500461854392320-ahHE?utm_source=share&utm_medium=member_desktop

Practice 5 SQL Projects for FREE…

https://www.linkedin.com/posts/ajay026_dataanalytics-dataanalysis-data-activity-7074230041861099520-Fc02?utm_source=share&utm_medium=member_desktop

Follow me here on LinkedIn for More resources….

https://www.linkedin.com/in/ajay026/